THE CGNS SYSTEM

Diane Poirier*
ICEM CFD Engineering, Berkeley, CA 94705
Steven R. Allmaras†, Douglas R. McCarthy‡, Matthew F. Smith§
Boeing Commercial Airplane Group, Seattle, WA 98124
and
Francis Y. Enomoto¶
NASA Ames Research Center, Moffett Field, CA 94035

The CGNS system consists of a collection of conventions, and conforming software, for the storage and retrieval of Computational Fluid Dynamics (CFD) data. It facilitates the exchange of data between sites and applications, and helps stabilize the archiving of aerodynamic data. The data are stored in a compact, binary format and are accessible through a complete and extensible library of functions. This API (Application Program Interface) is platform independent and can be easily implemented in C, C++, Fortran and Fortran90 applications. The CGNS system supports structured, unstructured and mixed topology, where multi-block connectivity may be either one-to-one abutting, mismatched abutting or overset. It defines standards for the storage of grid coordinates, flow solutions, boundary conditions, convergence history, reference state and geometry data. Dimensional units and nondimensionalization information may be associated with each type of data. Additionally, it provides conventions for archiving the governing equations including the gas, viscosity, thermal conductivity, turbulence and diffusion models. The CGNS system can be extended to other types of engineering analysis data, and serve multi-disciplinary applications. It is offered to the CFD community for the purpose of establishing a standard for aerodynamic data storage. This paper presents the different components of the CGNS system, from the essence of its constituents to its supporting data structures and software capacity. It demonstrates the facility to implement the CGNS system through a series of short examples, followed by a review of its incorporation into both research and commercial CFD applications.

1. Introduction

A series of meetings, held over approximately two years between airframe manufacturers and the government research community, addressed the improvement of means for transferring NASA technology to industrial use. It was held that a principal impediment to technology transfer was the widely disparate data-handling mechanisms endemic to the code development process. It is unfortunately too common to see the same CFD data set translated into various formats to perform the different tasks related to CFD analysis. Applications such as grid generation, flow computation, post processing or data visualization often use their own archiving system. As a result, different copies of the same data must coexist to insure compatibility with the different software tools. These data sets hold the same information expressed in different data structure systems and formats. Not only does this multiplicity of the data set waste storage media capacity and CPU time, but it also generates an enormous overhead in terms of data translator development, additional software and data management, customization of pre- and post- processors, etc. When integrated over the entire industry, these extra efforts reduce the cost effectiveness of CFD while impairing its development.

The "CFD General Notation System" (CGNS) was conceived to provide a general, portable and extensible standard for recording and recovering analysis data associated with the numerical solution of fluid dynamic equations.[1] It offers the opportunity for seamless communication between sites, platforms, and applications. By improving the interoperability of existing and future CFD tools, the CGNS system allows new software development to focus on functionality and reliability. It should therefore lead to the development of shared, reusable software selected on technical merit without concern for I/O compatibility.

The principal target of the CGNS system is the data normally associated with compressible viscous flow (i.e. the Navier-Stokes equations), but the standard is also applicable to simplified models such as Euler and potential flows. Much of the standard and conformed software utilities are applicable to computational field physics in general. Disciplines other than fluid dynamics would need to augment the data definitions and storage conventions, but the fundamental database software, which provides platform independence, is not specific to fluid dynamics.

The CGNS conventions and software provide for the recording of an extremely complete and flexible problem

* Software Engineer, Member AIAA.
† Principal Engineer, Senior Member AIAA.
‡ Principal Engineer.
§ Principal Engineer.
¶ Aerospace Engineer, Member AIAA.

description. A database may contain any number of structured and/or unstructured zones. These zones are described with their grid coordinates and connectivity information. Three types of multi-block connectivity are supported; overset (chimera), one-to-one abutting (point matched), and mismatched abutting. For unstructured zones, the element connectivity can be stored for a wide range of linear and higher order element shapes.

The mesh data is linked to the CAD data within the CGNS system to facilitate quick re-meshing after design changes or mesh optimization. The flow solutions may be defined at the vertices, or at cell, face or edge centers. Solution vectors are stored using precise naming conventions. Any number of flow variables may be recorded, with or without use of the standardized names.

The boundary condition specifications were developed to combine simplicity of initial implementation and generality to facilitate future extensions. They define boundary condition types, which establish the equations to be enforced. Dirichlet or Neumann boundary condition data may additionally be specified using CGNS conventions for data-name identifiers. The boundary condition specifications are general and flexible enough to provide for future extensions.

The CGNS system also provides for the storage of several types of auxiliary data. This includes the conventions for archiving the governing flow equations, the reference state quantities, the convergence history information, generic discrete or integral data, dimensional units and exponents, and nondimensionalization information. User's comments or documentation may be appended nearly anywhere, and it is also possible to add user defined or site specific data.

Not all of these data need to be present in the CGNS database at any particular time. The overall view is that of a shared database accessible by various software tools common to CFD: flow solvers, grid generators, field visualizers, post-processors, and so on. Each of these applications serves as an editor of the data, reading, adding to or modifying it according to that application's specific role.

Because of its generality, CGNS provides for the recording of much more descriptive information than current applications normally use. However, the provisions for these data are layered so that much of it is optional. It should be practical to convert most current applications to the CGNS system with little or no conceptual change, retaining the option to take advantage of more elaborate description as that becomes desirable.

A CGNS database may be written over any number of data files. Efficient internal linking allows the partitioning of the data over several files without reducing the performance of the data exchange. The individual files are more portable due to their reduced size, and enhance the functionality of the system. For example, the surface mesh may be kept separately from the field mesh, or several solutions may be linked to the same grid definition. Various mesh configurations can be combined instantly since the different databases need not be merged into a single file, but may be simply referenced. This flexibility facilitates parametric studies where various sub-domains may be automatically cycled over, and eases the management of interchangeable parts.

The CGNS system may be implemented in any CFD applications through the use of a complete and self-describing set of functions. This API is accessible through C or Fortran77 function calls allowing implementation in C, C++, Fortran77 and Fortran90 applications. The databases themselves are stored in compact C binary files. They are made machine independent through internal byte ordering translations, performed as needed and invisible to the user or application. The API performs extensive error checking on the database and informs the user of any irregularities via precise error diagnosis messages.

Currently the CGNS system is available on most architecture commonly used for CFD analysis: Cray/Unicos, SUN/Solaris, SGI/IRIX, IBM/AIX, HP/UX, DEC-Alpha/OSF. Windows NT support, on Dec and Intel platforms, is planned in the near future. Several applications, in the CFD research community and in industry, have already incorporated the CGNS standard successfully.

This paper describes the different elements of the CGNS system and presents examples of its implementation. Section 2 is divided into several sub-sections, each describing an individual component of the CGNS system. The hierarchical data structure “Advanced Data Format” (ADF) used to archive the databases is first presented. It is supported by a data exchange library called ADF Core, which comprises a set of low level routines for I/O operations on an ADF file. Following the presentation of the ADF and ADF Core, the “Standard Interface Data Structures” (SIDS) are introduced. The SIDS specification constitutes the soul of the CGNS system. Not only does it define the CGNS data structure, but it also establishes precisely the intellectual content of CFD-related data and prescribes conventions and nomenclature to standardize its archiving. Once the two basic elements of the CGNS system are described, the ADF and the SIDS, the next sub-section explains how to combine them together to form the CGNS system. This process is referred to as the “SIDS-to-ADF File Mapping”. The final element of the system is the CGNS mid-level library of software functions. It integrates the standards established by the other constituents of the

CGNS system into a user friendly API. The CGNS library is designed to facilitate the implementation of CGNS into any CFD applications. Section 3 demonstrates the implementation of the CGNS system through a series of short examples, and presents an overview of commercial and research CFD tools currently supporting the CGNS system. Finally, this paper concludes with a calendar of public releases planned for 1998, followed by recommendations for future enhancements.

2. The Elements of the CGNS System

2.1 The Data Structure ADF

The "Advanced Data Format" (ADF) is a concept defining how the data is organized in the storage media.[3] It is based on a single data structure called an ADF node, designed to store any type of data. Each ADF file is composed of at least one node called the "root". The ADF nodes follow a hierarchical arrangement from the root node down, as shown in figure 1.

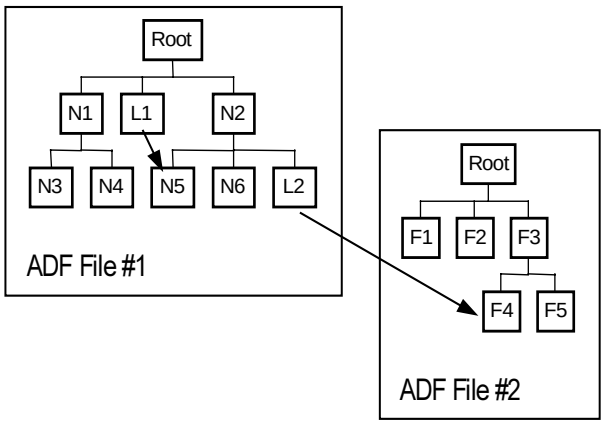

Fig.1 ADF Hierarchy of Nodes

The ADF node structure is composed of the following information:

- ID: A unique identifier to access a node within a file.
- Name: A character field used to name the node. It must be unique for a given parent.
- Label: A character field used to indicate the type of data contained in the node.
- Number of sub-nodes: The number of children directly attached to a node.
- Names of sub-nodes: The list of children names.
- Data type: A character field specifying the type of data (e.g. real, complex, character) associated with the node.
- Number of dimensions: The dimensionality of the data.
- Dimensions: An integer vector containing the number of elements within each dimension.
- Data: The data associated with the node.

An ADF data structure is a directed graph. Since the nodes hold the information about their children but ignore the identity of their parents, the hierarchy may only be traversed in one direction. The root node points to its children, which in turn point to theirs, and so forth. This simple pattern is repeated throughout the whole file resulting in a flexible hierarchical structure. There is no restriction on the number of children that a node can have, or on the number of levels in the hierarchy.

A node's children may be defined within the same ADF file, or in a different one. ADF supports the linking of ADF nodes stored in any ADF files. Since an ADF node may point directly to a child node located in another file, there is no performance penalty in using this feature. Internal and external links are illustrated in figure 1. The node L1 is linked to the node N5 within the same data file whereas the node L2 points to the node F4 located in a different ADF file. An ADF database is defined as the complete tree associated with a single ADF root node; this database may encompass several ADF files through the use of links.

This hierarchical structure is particularly suited for the storage of CFD databases, which are typically composed of a small number of very large arrays. A tree structure may be quickly traversed and sorted without the need of processing irrelevant information. The data exchange may therefore concentrate on the targeted arrays, resulting in improved performance. Although especially appropriate for the management of CFD-related information, the ADF is general and can be used to archive any type of data.

The ADF file has a header section that contains information about the file itself, such as the ADF library version used to create and modify the file, the date and time of creation and modification, and the data format used in the file (IEEE Big or Little Endian, Cray, etc.).

2.2. ADF Supporting Software: The ADF Core

The ADF Core is a library of low level I/O subroutines designed to implement the ADF concept.[3] The ADF format and library were developed as part of the CGNS project after examination of several data systems. These include the "Hierarchical Data Format" (HDF) developed at the National Center for Supercomputing Applications (NCSA) at the University of Illinois, the "Common File Format"

(CFF) created at McDonnell-Douglas Corporation (now Boeing St-Louis), and the "Network Common Data Form" (netCDF) sponsored by the National Science Foundation.[7,8,9] Although the existing systems present interesting features, none provide an appropriate hierarchical structure while affording the portability and extensibility of C software. It was therefore preferable to develop the ADF system.

The ADF Core is written in ANSI C to enhance the portability of the software, but provides a complete Fortran interface. It enables construction and browsing of new or existing ADF tree structures. The ADF Core is composed of 34 functions performing the following operations:

- open, close or delete and ADF file
- read or set the data binary format
- get the root-id or a node-id
- create, delete or move a node
- create, read or test a node link
- get the children of a node
- read or write the constituents of a node: name, label, data type, dimension, dimension vector and data
- perform version and error control

An ADF database is self-describing in the sense that it is not necessary to know its contents in order to read it. Using the ADF Core, one can easily browse through the hierarchy of an ADF database to reveal its constituents.

The databases are stored in compact C binary format. Each ADF node data field is characterized with a data type and the dimension of the data array. The supported data types are integer 32/64, unsigned integer 32/64, real 32/64, complex 64/128, character, byte and link. The ADF Core uses its own notation convention to identify the different data types independently of the system architecture. Table 1 lists the supported data types with their corresponding notation and machine representations. A given data type notation results in different binary representations depending on the system architecture.

While most of the supported platforms use the IEEE Big Endian standard, the Intel-Paragon and DEC-Alpha elected the IEEE Little Endian numeric format. Additionally binary files may be written using either a 32 or a 64-bit representation. These differences in the architecture native formats are resolved within the ADF Core insuring machine independence. An ADF file keeps track of the data format and operating system used at its creation. Whenever a binary format translation is necessary, the ADF Core executes it automatically. This is accomplished internally - without the need of any user intervention.

Type	Notation	Machine representation				
		Big Endian		Little Endian		Cray
		32bit	64bit	32bit	64bit	
no data	MT	-	-	-	-	-
integer-32	I4	I4	I4	I4	I4	I8
integer-64	I8	-	I8	-	I8	I8
unsigned int-32	U4	I4	I4	I4	I4	I8
unsigned int-64	U8	-	I8	-	I8	I8
real-32	R4	R4	R4	R4	R4	R8
real-64	R8	R8	R8	R8	R8	R8
complex 64	X4	R4R4	R4R4	R4R4	R4R4	R8R8
complex 128	X8	R8R8	R8R8	R8R8	R8R8	R8R8
character	C1	C1	C1	C1	C1	C1
byte	C1	C1	C1	C1	C1	C1

Table 1. ADF Supported Data Types

When opening a new ADF file, it is also possible to choose the binary representation independently of the system architecture. For example, an ADF file may be written in the Cray native format from an SGI running IRIX (or vice versa). If unspecified, the ADF Core uses the local architecture native format by default.

The ADF Core has been tested and used on several platforms, namely Cray/Unicos, SUN/Solaris, SGI/IRIX, IBM/AIX, HP/UX, DEC-Alpha/OSF and Intel-Paragon. Aside from CGNS, the ADF Core has also been adopted as the underlying data structure for the latest release of the "Common File Format" (CFF), which is used by the NPARC Alliance's code WIND.[10,11] CFD codes could use the ADF Core directly, but have the advantage of higher level routines provided by the CGNS library (section 2.4); this library is built on top of the ADF Core. The ADF Core software and documentation are available at http://www.cgns.org.

2.3. The Standard Interface Data Structures, SIDS

The "Standard Interface Data Structures" specification constitutes the essence of the CGNS system. While the other elements of the system deal with software implementation issues, the SIDS specification concerns itself with defining the substance of CGNS. It precisely defines the intellectual content of CFD-related data, including the organizational structure supporting such data and the conventions adopted to standardize the data exchange process.[2]

The SIDS are designed to support all types of information involved in CFD analysis. While the initial target was to establish a standard for 3D structured multi-block compressible Navier-Stokes analysis, the SIDS extensible framework now includes unstructured analysis, 2D

configurations, hybrid topology and geometry-to-mesh association. Although the SIDS specification is independent of the physical file formats, its design was targeted towards implementation using the ADF Core library. Some of the language components used to define the SIDS are meant to directly map into elements of an ADF node. Furthermore, the data structures specified in the SIDS are organized in a hierarchical manner in accordance with the ADF topology.

The data sets typical of CFD analysis tend to contain a small number of extremely large data arrays. This implies that the I/O system must efficiently store and process large data arrays. The SIDS are designed to optimize the performance of the data exchange process supported by the ADF Core. A second implication of the nature of the data resides in the opportunity to include thorough description in the file with relatively little storage overhead and performance penalty. For example, the flow solution of a CFD analysis may contain several millions values. Therefore, with little overhead, it is possible to include information describing the flow variables stored, their location in the grid, and the dimensional units or nondimensionalization information associated with the data. The SIDS specification takes advantage of this situation and includes an extensive description of the information contained in its data structures.

Other design considerations were the minimization of duplicated data within the hierarchy and the ability to include documentation throughout the database. Whenever possible, generic data structures were developed to hold various types of CFD information. On the other hand, consistency dictated the development of specialized data structures for certain types of CFD-related information.

The SIDS conventions provide for the recording of an extremely complete and flexible problem description. This section gives an overview of the main data structures defined in the SIDS, as well as some examples of the standardized nomenclature. It demonstrates the vast range of CFD analysis data covered by this standard, and the explicitness in which the data can be archived using the SIDS conventions. It is important to keep in mind, while reading the next paragraphs, that all of these data need not be present. The SIDS are layered so that much of its data structures are optional.

In the following sub-sections, standard SIDS names and identifiers are differentiated from the regular text by the use of a different character font. Most data structure names carry the suffix `_t` (for type) to distinguish them from regular data.

2.3.1 CGNS Base Data Structure: `CGNSBase_t`

The data structure at the root of the CGNS tree graph is called `CGNSBase_t`. It is illustrated in figure 2. It contains

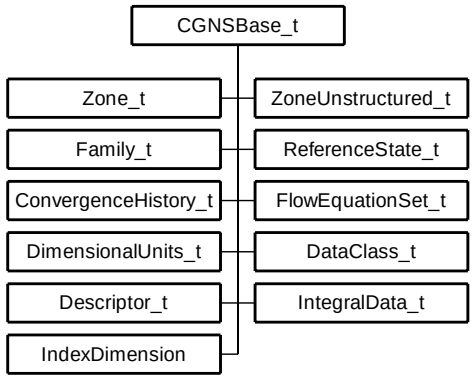

Fig.2 CGNS High Levels Chart

the dimensionality of the computational grid (`IndexDimension`) and several sub-structures such as the zones (structured or unstructured blocks) constituting the CFD model. The `CGNSbase_t` includes also the family sub-structures where geometry-to-mesh associations are recorded. Additionally, auxiliary information applicable to the entire `CGNSBase_t` data structure may be stored at this level. This includes the reference state data, dimensional units, nondimensionalization information (`DataClass_t`), flow equation sets, documentation (`Descriptor_t`) and convergence history data structures. The dimensionality of the computational grid (`IndexDimension`) is the sole mandatory element of this data structure. It is defined as the number of indices needed to uniquely identify a vertex within the grid.

2.3.2 Zone: `Zone_t` or `ZoneUnstructured_t`

The zone data structure contains all the information pertinent to an individual zone or grid block within the domain. Two types of zones are defined in the SIDS, `ZoneUnstructured_t` and `Zone_t` for unstructured and structured mesh block respectively. For both structured and unstructured blocks, the only mandatory elements of the zone data structure are the number of cells and vertices contained in the zone. A zone may optionally contain sub-structures defining the physical coordinates of the computational grid, the flow solutions, the interface connectivity and the boundary conditions. Additionally, auxiliary information applicable to the entire zone may be stored at this level. This includes the data structures defined for reference state data, nondimensionalization information, dimensional units, flow equations set, documentation and zone convergence history. Figure 3 illustrates the constituents of a structured zone data structure. Unstructured zones contain one additional data structure for the definition of the element connectivity data.

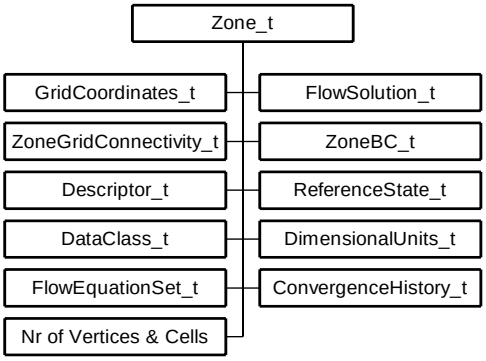

Fig.3 Zone Data Structure

2.3.3 Grid Coordinates: `GridCoordinates_t`

The physical coordinates of the computational grid are defined by the grid coordinates data structure, as shown in figure 4. This structure contains a list of data arrays

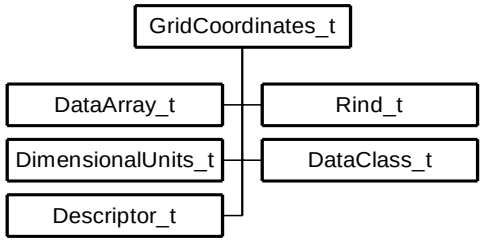

Fig.4 Grid Coordinates Data Structure

representing the individual components of the position vector. It also provides a mechanism for identifying rind-point data (dummy layers) included within the position vector arrays. If necessary, the nondimensionalization information and dimensional unit sub-structures may also be defined.

The SIDS support coordinate definition in Cartesian, cylindrical and spherical coordinate systems. In addition, it also provide the means to define local (auxiliary) coordinate systems, often used to define normal or tangential stresses. A series of standardized names supplied by the SIDS unambiguously identifies the content of each coordinate data array. These data-name identifiers are self-describing, for example `CoordinateX` and `CoordinatePhi`.

2.3.4 Flow Solution: `FlowSolution_t`

The flow solution data structure, `FlowSolution_t`, is used to record one solution data set. There is no limit on the number of solutions sets contained in a zone data structure. Each solution set, in term, may include one to several solution vectors. A flow solution data structure could be used, for example, to hold the initial or restart solution, while a second one could serve to record the computed solution after some number of iterations.

The flow solution data structure contains all the sub-structures found in the grid coordinate structure (data arrays, dimensional information, rind-data and documentation), with additionally the grid location parameter. Unlike the grid coordinates, which always coincide with the vertices, the flow solutions may be defined at the vertices, or at cell, face or edge centers. This extra feature necessitates the creation of two different data structures to hold grid coordinates and flow solutions. Once again, the SIDS specification regulates the variable names in order to facilitate a standardized data exchange. A list of data-name identifiers for typical Navier-Stokes solution variables was established, and can be easily extended if needed. It contains identifiers such as `Pressure`, `VelocityX`, `SkinFrictionY`, `MassFlow`, etc. It is also possible to record any type of site specific variables, even if they are not included in the SIDS nomenclature.

2.3.5 Zone connectivity: `ZoneGridConnectivity_t`

The zone connectivity information may be recorded for each zone under a general data structure called `ZoneGridConnectivity_t`. This data structure is illustrated in figure 5. It holds three sub-structures responsible for the grid connectivity data, `GridConnectivity_t`, `GridConnectivity1to1_t`, and `OversetHoles_t`. It may also include some documentation recorded in the sub-structure `Descriptor_t`.

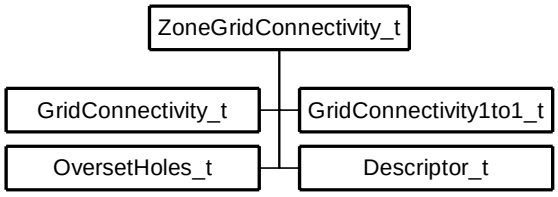

Fig.5 Zone Grid Connectivity Data Structure

All three types of multi-block connectivity may be defined using the general connectivity sub-structure called `GridConnectivity_t`. It contains the list or range of indices defining the interface in the current zone (receiver), the name of the adjacent zone (donor), the list of points on the donor side and a parameter specifying the type of connectivity.

If the interface is constituted of points having consecutive indices, the patch may be defined by simply referencing the first and last indices of the range. This patch definition method is called `PointRange`. When the points do not have consecutive indices numbering, they are all recorded in a structure called `PointList`. For example, an abutting one-to-one interface between two structured blocks represents a rectangular sub-range in the computational domain. It can be simply defined using a `PointRange`.

For overset interface, the points on the receiver side identify the fringe points outlining the overset hole. These are most likely non-consecutive indices requiring the use of a `PointList` representation.

The list of points on the donor side, `PointListDonor`, contains the images of the receiver zone interface points in the donor zone. These are real values identifying the bi- or tri-linear interpolation factors used to define the location of each receiver point in the donor zone grid. For the particular case where the points on the donor side coincide with those on the receiver side, the donor points correspond to the indices of the nodes on the donor side. If the donor zone is unstructured, each receiver node is linked to an element of the unstructured block, in addition to the interpolation factors. The types of connectivity are identified using the standardized names `Overset`, `Abutting` and `Abutting1to1`.

A special sub-structure is defined in the SIDS for the recording of an abutting interface patch with one-to-one point matching between two adjacent structured zones. In this particular case the connectivity may be entirely defined by simply identifying the range of points delimiting the interface in both adjacent zones, and a transformation matrix describing the relative indices orientation of the zones. This information is recorded in `GridConnectivity1to1_t`, a special data structure created for this particular case.

The grid connectivity information for overset grids must also account for the overset holes within each zone. These holes identify regions where the flow solution is ignored since it is being solved in some other overlapping zone. The data structure `OversetHoles_t`, located also within the `ZoneGridConnectivity_t` structure, holds the definition of the holes within a zone. The `OversetHoles_t` data structure provides for the recording of a `PointList` or a list of `PointRange`, any relevant documentation and the grid location (vertex or cell center) referenced by the point indices.

2.3.6 Zonal Boundary Condition: `ZoneBC_t`

The boundary conditions can be defined either on mesh patches or on geometrical entities (explained in section 2.3.12). Associating boundary conditions to mesh patches permits specification of local data sets at the vertices or face centers of the boundary condition patch. The zonal boundary condition structure, `ZoneBC_t`, also provides a means to define boundary conditions without requiring geometric data in the file.

Within the hierarchy, the `ZoneBC_t` structure is located directly under each zone and contains the list of boundary condition structures (`BC_t`) pertaining to the zone. Each `BC_t` sub-structure contains the boundary condition information for a single patch of the zone. It provides for the recording of a boundary condition type (`BCType_t`) as well as one or more sets of boundary condition data (`BCDataSet_t`). The `BC_t` data structure also contains information describing the patch itself, such as its location (`PointRange` or `PointList`) and its normal vector definition. Figure 6 illustrates the sub-structures contain in the zone boundary condition data structure, `ZoneBC_t`. The generic term "Auxiliary Data" is used in place of the sub-structures `ReferenceState_t`, `DimensionalUnits_t`, `DataClass_t`, and `Descriptor_t`.

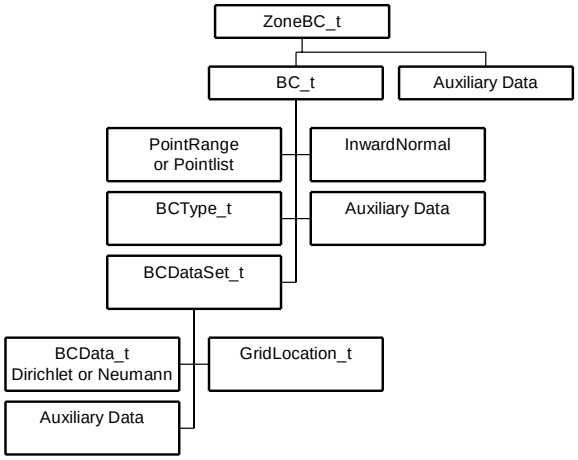

Fig.6 Zone Boundary Conditions Data Structure

The first item identifying the boundary condition equations to be enforced at a given boundary location is the boundary condition type, `BCType_t`. The boundary condition types are subdivided in two categories: `BCTypeSimple_t` and `BCTypeCompound_t`. For simple boundary conditions, the equations and data imposed are fixed, whereas for compound boundary conditions, different sets of equations are imposed depending on local flow conditions at the boundary. The boundary condition types are identified using standardized names such as `BCDirichlet`, `BCWallViscous` and `BCInflowSubsonic`. The second item used in the definition of a boundary condition is the boundary condition data set, `BCDataSet_t`. It holds a list of variables defining the boundary condition. Each variable may be given as global data or local data defined at each grid point of the boundary condition patch.

The coupling of a boundary condition type and a boundary condition data set allows formation of the governing equations at the boundary. For example, the boundary condition type `BCNeumann` indicates a Neumann condition $\delta Q/\delta n$ at the boundary. Here Q stands for the solution vector and n for the normal to the boundary. The precise equation is then built using the boundary condition solution data specified in the `BCDataSet_t`:

$$\delta Q/\delta n = (\delta Q/\delta n)_{specified}$$

2.3.7 Flow Equation Set: `FlowEquationSet_t`

The `FlowEquationSet_t` data structure is used for recording a general description of the governing flow equations. This data structure may be included in the `CGNSBase_t` structure or at the zone level, depending if the equations defined are applied to the entire configuration or to a specific zone.

The flow equation set data structure was designed to balance the opposing requirements of extensibility for future growth and initial ease of implementation. It is intended primarily for archival purposes, providing additional documentation of the flow solution. However, it is foreseeable that these flow equation structures may also serve as inputs for grid generators, flow solvers, and post processors.

The `FlowEquationSet_t` data structure provides for the storage of the general class of governing equations, the gas, viscosity, thermal conductivity and turbulence models, the turbulent closure equation, and the dimensionality of the governing equations. Each of these equations or models forms a sub-structure of the `FlowEquationSet_t` structure. The flow equation set data structure is illustrated in figure 7.

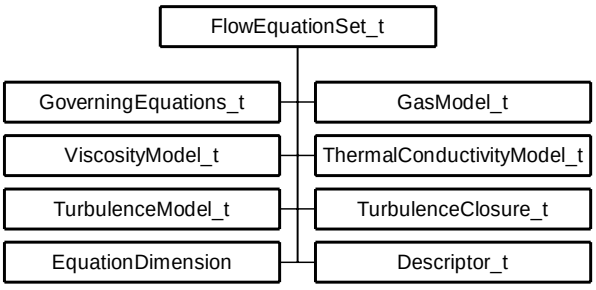

Fig.7 Flow Equation Set Data Structure

The governing equation class currently supported include full potential, Euler, Navier-Stokes laminar, Navier-Stokes turbulent, Navier-Stokes laminar incompressible and Navier-Stokes turbulent incompressible. When the Navier-Stokes equations are used, the governing equation sub-structure (`GoverningEquations_t`) provides for the recording of the diffusion terms modeled in the flow equations.

The thermodynamic gas model data structure, `GasModel_t`, specifies the equation of state used in the governing equations to relate pressure, temperature and density. Two model types are supported, `Ideal` and `VanderWaals`. This data structure also allows for the archiving of related quantities such as the ideal gas constant (R) or the specific heat at constant pressure or volume (c_p, c_v). These are recorded using the SIDS standardized name identifiers `IdealGasConstant`, `SpecificHeatVolume`, etc.

The molecular viscosity model structure, `ViscosityModel_t`, specifies the model used for relating molecular viscosity (μ) to temperature. It supports constant, power law and Sutherland's Law models, as well as their related quantities.

The `ThermalConductivityModel_t` structure specifies the model used for relating the thermal conductivity coefficient (k) to the temperature. The SIDS support the constant Prandtl number ($Pr = \mu\, c_p/k$) case, power law and the Sutherland's Law, with their related quantities.

The `TurbulenceClosure_t` structure describes the turbulence closure for the Reynolds stress terms of the Reynolds-averaged Navier-Stokes equations. The types supported are `EddyViscosity`, `ReynoldsStress` and `ReynoldsStressAlgebraic`. The SIDS support turbulence models, such as `Algebraic_Baldwin-Lomax` or `OneEquation_Spalart-Allmaras`, for example. Details on turbulence closure and modeling can be found in reference 2.

2.3.8 Reference State: `ReferenceState_t`

The `ReferenceState_t` data structure contains a set of reference state flow conditions defined at a reference location or condition. The use of data-name identifiers allows once more the standardization of the data. The `ReferenceState_t` structure holds the definition of flow state quantities such as `VelocitySound`, `Temperature`, `PressureStagnation`, etc. It also allows for the storage of the dimensional units and any related documentation.

2.3.9 Data Class and Conversion

`DataClass_t` identifies the class of a given piece of data. These classes divide data into different categories depending on dimensional units and normalization associated with the data. The data class called `Dimensional` specifies dimensional data. Nondimensional data that is normalized by dimensional reference quantities are included in the data class `NormalizedByDimensional`. In contrast, `NormalizedByArbitraryDimensional` specifies non-dimensional data typically found in completely nondimensional databases, where all fields and reference data are nondimensional. `NondimensionalParameter` indicates nondimensional parameters such as the Mach number and the lift coefficient. Constants such as π are designated by the data class `DimensionlessConstant`.

The `DataConversion_t` data structure contains conversion factors for recovering raw dimensional data from given nondimensional data. These conversion factors are typically associated with nondimensional data that is normalized by dimensional reference quantities (class `NormalizedByDimensional`).

2.3.10 Dimensional Units and Exponents

The data structure `DimensionalUnits_t` describes the system of units used to measure dimensional data. It is composed of a set of types that define the mass, length, time, temperature and angle units.

The dimensionality of the data is described by defining the exponents associated with each of the fundamental units, i.e. mass, length, time, temperature and angle. The dimensional exponents are recorded in the data structure `DimensionalExponents_t`.

2.3.11 Precedence Rule Within the Hierarchy

A few types of data structures defined in the SIDS may be recorded at several levels of the hierarchy. These include entities for describing data class, system of dimensional units, reference states and flow equation sets. The precedence rule established by the SIDS states that if such structures are present at one level, they take precedence over all corresponding information existing at higher levels of the CGNS hierarchy. Essentially, the SIDS specification establishes globally applicable data with provisions for recursively overriding them with local data.

The `ReferenceState_t` data structure for example, may be defined within a `CGNSBase_t` structure, a `Zone_t` or `ZoneUnstructured_t` structure, or at several levels of the boundary condition hierarchy. If it is defined simultaneously within a `CGNSBase_t` data structure, and a `Zone_t` contained in `CGNSBase_t`, the reference data defined for the zone supersedes the global definition within that zone only. This relationship is displayed on figure 8.

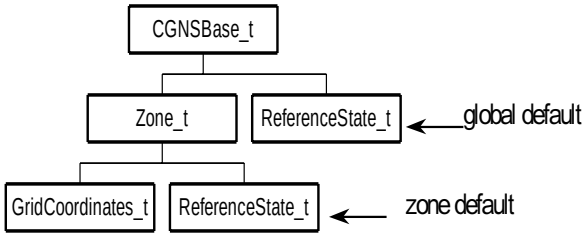

Fig.8 Globally Applicable Data and Precedence

2.3.12 Family Data Structure: `Family_t`

The `Family_t` data structure connects the geometry data of the various components of a model to the computational grid in such a way that given a mesh surface, the underlying geometry can be determined, or vice versa. The geometry-to-grid connectivity is defined by associating node or cell regions to geometric entities described within a given CAD data file. The SIDS specification does not define a new standard for the storage of CAD data, but rather establishes conventions for referencing geometric entities stored in a CAD database.

There is rarely a one-to-one connection between mesh regions and geometric entities. Consequently, the mesh-geometry associations make use of a layer of indirection. Rather than linking the geometry data to the mesh entities (nodes, edges and faces), these data are associated with intermediate objects. The intermediate objects are in turn linked to the nodal regions of the computational mesh. These intermediate objects are called CFD families. Node and family association is implemented by assigning a family name to each boundary condition patch of the mesh zones. These family names serve as pointers to the various `Family_t` sub-structures bearing the same names.

The `Family_t` data structure is illustrated in figure 9. It contains two main sub-structures, `GeometryReference_t` and `FamilyBC_t`. It may also contain any related documentation.

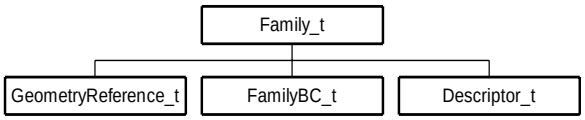

Fig.9 Family Data Structure

The `GeometryReference_t` data structure identifies the CAD systems used to generate the geometry, the CAD files where the geometry is stored and the list of CAD entities within these files corresponding to the given family. There is no restriction on the CAD system as long as it supports CAD entity attributes, used as handles in the referencing process. A mesh may be associated to any number of CAD files, which may encompass several CAD systems. It is also possible to use directly the CAD entity names to link the mesh to the geometry. In such case, the family names are set to the CAD attributes and the list of CAD entity attributes in the `GeometryReference_t` data structure is left blank.

The `FamilyBC_t` data structure provides an alternative for the definition of boundary conditions. As discussed earlier in section 2.3.6, the boundary conditions may be defined on mesh patches under each zone structure. This has the advantage of providing a means for storing mesh related flow solution data. The other option for specifying boundary condition is to link them to the CFD families. When both mesh patch boundary condition and family boundary condition are defined simultaneously for a same boundary, the definition attached to the mesh entity has precedence over the one defined for the geometric family.

The main advantage of associating the boundary conditions to the families is that the mesh topology or mesh density may be modified without altering the boundary condition settings. Another motivation for choosing this alternative

method is that any given boundary condition needs only to be defined once, even if it is applied to several patches over multiple zones. The `FamilyBC_t` structure includes the boundary condition type and various auxiliary data.

This concludes the overview of the SIDS. Reference 2 provides a comprehensive definition of the SIDS conventions, data structures and data-name identifiers. This information is also available at http://www.cgns.org.

2.4. The SIDS-to-ADF Mapping

As seen in the previous sections, the ADF and ADF Core define a new database format and its supporting software, while the SIDS specify precisely the contents of a CGNS archive. These two elements are combined to form the CGNS hierarchical database specification. This coupling of the ADF and SIDS is called "SIDS-to-ADF Mapping". It transfers the constituents of the SIDS to an underlying ADF structure. Each data structure defined in the SIDS is mapped to one or more ADF nodes, while maintaining the hierarchical organization of the SIDS. The result is called a CGNS database. It consists of a sub-tree of an ADF file or files rooted at a node labeled `CGNSBase_t`. It conforms to the SIDS model as implemented by the "SIDS-to-ADF Mapping" specification, thus may be accessed using the ADF Core library.

The "SIDS-to-ADF Mapping" specification associates each piece of information defined in the SIDS to a precise location in the ADF structure.[4] In most cases, the ADF node label holds the data structure type identifier as defined by the SIDS. For example, a zone defined using the data structure `Zone_t` or `ZoneUnstructured_t` in the SIDS would be associated with an ADF node labeled `Zone_t` or `ZoneUnstructured_t` in the CGNS database.

The names of the children nodes must be unique for any given parent. For example, a CGNS database contains as many children of type `Zone_t` as there are structured grid zones in the domain; each zone must have a distinct name. By convention, some ADF node names within the CGNS hierarchy are fixed. However most node names can be specified by the user. The "SIDS-to-ADF Mapping" specification states that the supporting software must provide for default naming capability. The default names are constructed by replacing the label last two characters (`_t`) with a positive integer. According to this convention, the names for *N* structured zones under the same `CGNSBase_t` data structure are *Zone1, Zone2, Zone3, …, ZoneN*.

The entry structure of the SIDS, called the `CGNSBase_t`, is mapped to an ADF node labeled `CGNSBase_t`. Figures 3 and 10 illustrate the one-to-one correspondence between the SIDS and the "SIDS-to-ADF Mapping". There may be one

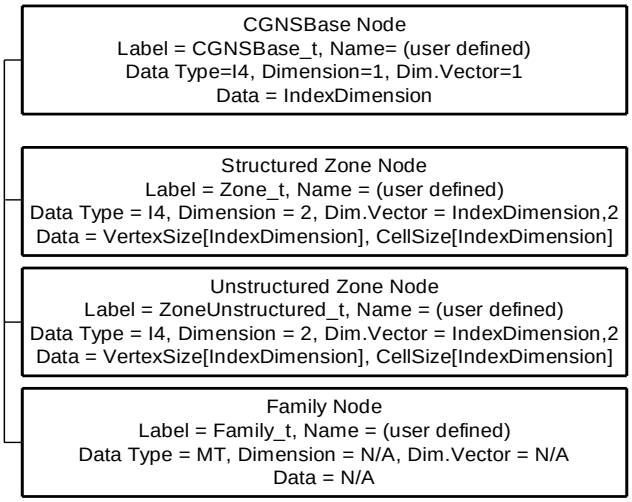

Fig.10 SIDS-to-ADF Mapping of Upper Levels

or many `CGNSBase_t` nodes in a CGNS file. The index dimension (`IndexDimension`) of the computational space is stored in the data field of the `CGNSBase_t` node. The data type is therefore integer, specifically "I4" (using the ADF nomenclature). In this case, the dimension and dimension vector are both equal to one.

The `CGNSBase_t` node may have several types of children. The data structure for a single block structured zone, located directly under the `CGNSBase_t` node, is an ADF node labeled `Zone_t`. The data field for `Zone_t` holds the number of vertices and cells within each dimension of the computational domain. The data type is I4, the dimension of the data array is 2, and the dimension vector is (`IndexDimension`, 2). For a structured 3D zone, this translates into a 3×2 array of integers. The first three values express the number of vertices within each dimension, while the last three contain the number of cells.

Each `Zone_t` node may contain one `GridCoordinates_t` node, one `ZoneGridConnectivity_t` node, one `ZoneBC_t` node, and one or several `FlowSolution_t` nodes as shown in figure 11. These nodes do not contain any data (data type = MT), but instead, open new branches of the ADF tree structure for storage of their respective data structures. Grid coordinates, for example, are stored using the generic node type `DataArray_t`. Each coordinate vector is contained into an ADF node labeled `DataArray_t` and located directly under the `GridCoordinates_t` node. The arrays of coordinate values carry the dimension specified by `IndexDimension`

and may be written in single or double precision (R4 or R8 data type). The dimension vector is a function of the vertex size and the number of rind-planes associated with the data array.

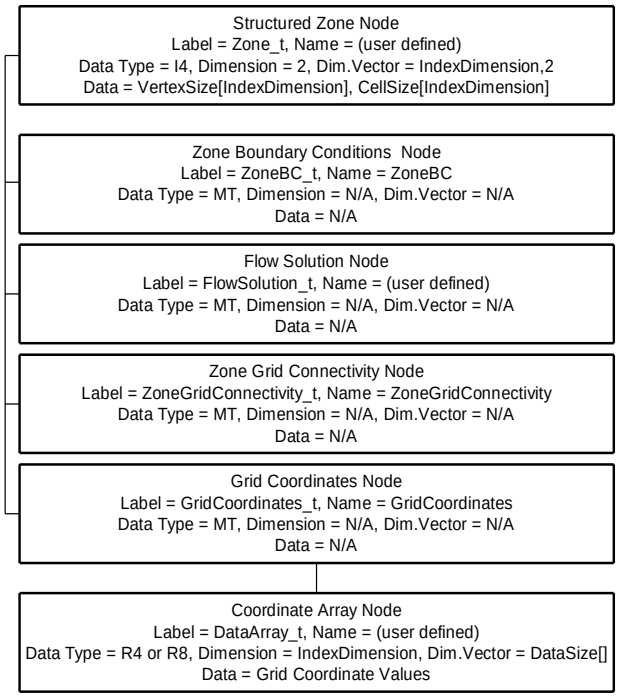

Fig.11 SIDS-to-ADF Mapping of a Zone

These few examples demonstrate the process followed to map the SIDS contents onto the ADF Structure. Reference 4 provides a complete description of the entire "SIDS-to-ADF Mapping" specification. This information is also available at http://www.cgns.org.

2.5 The CGNS Library

This section outlines the CGNS library, which was designed to ease the implementation of CGNS by providing developers with a collection of handy I/O functions.[5] Since knowledge of the ADF core is not required to use this library, it greatly facilitates the task of incorporating the CGNS system in any CFD applications.

The CGNS library is based on the SIDS and "SIDS-to-ADF Mapping" specifications, and built using the ADF Core. It allows reading and writing CGNS databases through the use of a user friendly API. The library is written in ANSI C to enhance its portability. However, each function has a Fortran77 interface counterpart to ease implementation in Fortran77 or Fortran90 applications. Each C routine name has two segments, first the prefix "`cg`" indicating the origin of the routine (CGNS API), and then a few words describing unambiguously its functionality. The Fortran interface names are built identically, with the addition of the suffix "`_f`" to distinguish them from the C function nomenclature. For example, a CGNS zone may be read using the C function `cg_zone_read` or the Fortran interface routine `cg_zone_read_f`.

The first step when accessing a CGNS file consists in opening the data exchange process. This is accomplished by the function `cg_open`. This routine opens a new or existing CGNS file, initializes the file if it is new, and set up the library internal data structures. An existing CGNS file may be opened to read or modify its contents, while a new one may only be opened for writing. One of the arguments of the function `cg_open` is therefore to specify the action or mode desired: `READ`, `WRITE` or `MODIFY`. The last step when completing the data exchange with a CGNS file is performed using the routine `cg_close`. This function updates the contents of the file stored on disk and terminates the dialogue.

When the file is opened with `cg_open`, in modes `READ` or `MODIFY`, the CGNS library parses the entire tree structure and loads most of its contents in memory. This internal representation of the data is stored in memory using C structures similar to those described in the SIDS. Similarly, when a CGNS file is opened with mode `WRITE`, the information transmitted by the CFD application is first accumulated in the internal data structure, and only written to the storage media when the data exchange is terminated using `cg_close`. The use of an internal data depiction affords the advantage of increasing the performance of subsequent I/O requests, since the information is readily available without requiring additional tree parsing. Another benefit stems from limiting the relatively slow communication to and from the storage media, since most of these data exchanges take place all at once, when calling `cg_open` or `cg_close`.

While `cg_open` and `cg_close` take care of the communication between the storage media and the library internal data representation, the other functions of the CGNS library handle the data exchange between the CFD applications and the memory depiction of the data. This is schematically demonstrated in Figure 12. This

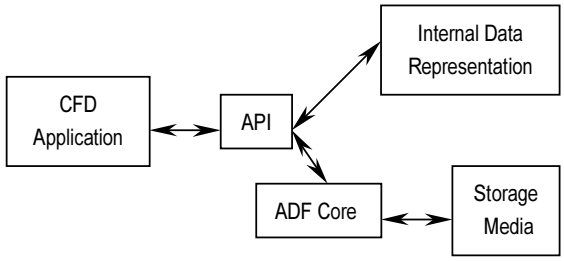

Fig.12 Data Flow

representation of the data flow has one exception. In order to reduce memory usage and improve execution speed, large arrays such as grid coordinates or flow solutions are not actually stored in memory. Instead, only their ADF ID numbers (addresses) are recorded in the internal data representation. If these large arrays are requested in subsequent function calls, the ADF addresses enable the software to locate them immediately on disk, without requiring a search through the tree structure. Likewise, when a CGNS file is opened in writing mode, the large arrays transmitted to the library are immediately written to disk. Since the hierarchical data structure holds only the root node at this point, the nodes containing the large data arrays are recorded directly under the ADF root node. When the file is closed, these arrays are moved (by reference only) to their appropriate location in the CGNS hierarchy.

The CGNS library was designed to mirror the structure of the SIDS. Each type of data structure defined in the SIDS is supported in the CGNS library by a set of reading and writing functions. For example, the data exchange for a structured zone is completely handled by the following three functions:

- `cg_nzones(…)` :read the number of zones.
- `cg_zone_read(…)` :read the zone information.
- `cg_zone_write(…)` :write a new zone.

Similar sets of functions exist for every data structure defined by the SIDS. The hierarchical organization of this API facilitates the implementation of CGNS in CFD applications, while insuring its extensibility for future development of the SIDS framework.

Most C functions of the CGNS library return an integer value representing the error status. The Fortran functions contain an additional argument, `ier`, which holds the value of the error status. An error status different from zero reports the occurrence of a problem during the execution of a function. Precise and concise error diagnosis may be printed using one of the error handling functions of the CGNS library:

- `cg_get_error()` :return the error message in a character field.
- `cg_error_print()` :get and print the error message.
- `cg_error_exit()` :get and print the error message, and terminate program execution.

The CGNS library defines variable types in conformance with the SIDS nomenclature. These facilitate the implementation of the CGNS API. The various boundary condition identifiers, for example, are part of an enumeration for the variable type `BCType_t`.

```
BCType_t = { BCInflowSupersonic,
             BCSymmetryPlane,…,
             BCWallViscous}
```

These specialized variable types are defined in the C include file `cgnslib.h` by an enumeration of keywords admissible for any variable of these types (`typedef enum`). This file must be included into any C application program using these data types. Similarly in Fortran, these keywords are defined as integer parameters in the include file `cgnslib_f.h`. This file must be included into any Fortran application using these keywords. The utilization of keywords affords the opportunity to work with SIDS name identifiers in CFD applications, without having to deal with character string variables. The identifiers `Overset` or `Abutting1to1` may be used integrally in the CFD application to identify an interface type. Likewise, the keywords `Vertex`, `CellCenter`, `FaceCenter` are meaningful to the library, when specified in a function's argument list, to describe a grid location. The CGNS library offers these lists of keywords as a convenience to the programmers using the API. Their utilization enhances the code visibility while facilitating variable declaration and memory allocation.

Another asset of the CGNS library is to directly retrieve the ADF ID number of any ADF node in the database. A simple function call reveals the ADF address of nodes such as `CGNSBase_t`, `Zone_t`, `GridCoordinates_t`, `ZoneBC_t`, etc. This feature helps implementing any type of site specific information, which may not yet be supported by the CGNS SIDS and API.

Site specific data may be included anywhere within the CGNS hierarchy without hindering the database compatibility with the CGNS API. In order to include site specific data structures, these must be simply recorded in ADF nodes tailored for their use, by means of the ADF Core. Since the CGNS API searches for specific node labels while ignoring the others, any addition of node types has no effect on its functionality. The CGNS library software and documentation are available at http://www.cgns.org.

This concludes the description of the CGNS elements and their relationship with one another. The next section demonstrates the implementation of CGNS by means of three small examples, which are thoroughly explained. Then it reviews the status of the CGNS system incorporation in various research and industrial CFD applications. Finally it presents the schedule of releases of the CGNS system's software and documentation, planned for 1998, and explains the differences between the release versions.

3. Implementation

Section 3.1 shows a few examples on how to use the CGNS library to read and write CGNS databases. In each example, the error checking, variable declarations and memory allocations were omitted to lighten the text. The parameters returned by each function are printed in bold to distinguish them from the input arguments.

3.1 Examples

3.1.1 Read a Multi-Block Structured Mesh

The following C program (figure 13) reads the X-coordinate vector of all structured zones in all the CGNSBase_t data structures included in a CGNS file. The functions and variables used in this example are described in detail in the following paragraphs.

```
cg_open(FileName, MODE_READ, &FileNo)
cg_nbases(FileNo, &NBases);
for (BaseNo=1; BaseNo<=NBases; BaseNo++){
  cg_base_read(FileNo, BaseNo, BaseName,
               &IndexDimension);
  cg_nzones(FileNo, BaseNo, &NZones);
  for (ZoneNo=1;ZoneNo<=NZones;ZoneNo++){
    cg_zone_read(FileNo, BaseNo, ZoneNo,
                 ZoneName, ZoneSize);
    cg_coord_read(FileNo, BaseNo, ZoneNo,
                 "CoordinateX",RealSingle,
                  RangeMin, RangeMax, X);
}}
cg_close(FileNo);
```

Fig.13 Example Reading Bases, Zones and Coordinates

In this first example, a CGNS file, identified by the character variable FileName, is opened for reading (mode = MODE_READ). Since several CGNS files may be opened simultaneously, the function cg_open returns a file number (FileNo). It consists of an integer value uniquely identifying a file in the CGNS library. The file number must be used in every subsequent function calls to specify which of the opened CGNS files must be accessed.

The next function, cg_nbases, returns the number of CGNSBase_t structures in the CGNS file (Nbases). There is no limit on the number of CGNSBase_t data structures contained in a file. Several databases could be used to record slightly different configurations of a same model for example. The CGNSBase_t information is read with the function cg_base_read. Given a file number and a database number (BaseNo), this routine returns the name of a database (BaseName) and the dimension of the computational domain (IndexDimension).

The number of zones (NZones) within each CGNSBase_t data structure is extracted using the function cg_nzones. The zone name (ZoneName) and size (ZoneSize) are read with the routine cg_zone_read for each zone number (ZoneNo). Note that the zone size is a vector of integers containing both the numbers of vertices and cells within each computational dimension.

The function cg_coord_read returns the location vector of a coordinate in the format and range requested. The coordinate requested is specified using its data-name identifier, in this case CoordinateX. The precision in which the data array must be provided to the CFD application may be set, regardless of the data type used to store this information on disk. The CGNS library handles the conversion if necessary. A data array recorded as double precision in the storage media may be read as single precision by the CFD application, or vice versa. This routine supports two data formats, RealSingle and RealDouble, for single and double precision values respectively. RealSingle and RealDouble are defined as keywords in the library include file. The data array range requested is specified by the integer vectors RangeMin and RangeMax. These vectors specify the minimum and maximum computational index value of the range requested, within each dimension.

3.1.2 Read the Solution Data of a Structured Zone

Figure 14 shows an example of a Fortran program reading the solution data contained in a zone. A zone may hold any number of flow solutions. These could be recorded at different time steps for example. Each flow solution may in turn contain any number flow variables. The functions and variables used in this example are detailed in the following paragraphs.

```
call cg_nsols_f(FileNo, BaseNo, ZoneNo,
                NSolutions, ier)
do SolutionNo=1, NSolutions
 call cg_sol_info_f(FileNo, BaseNo,
      ZoneNo, SolutionNo, SolutionName,
      GridLocation, ier)
 call cg_nfields_f(FileNo, BaseNo,
      ZoneNo, SolutionNo, NFields, ier)

 do FieldNo=1, NFields
  call cg_field_info_f(FileNo, BaseNo,
       ZoneNo, SolutionNo, FieldNo,
       DataType, FieldName, ier)
  call cg_field_read_f(FileNo, BaseNo,
       ZoneNo, SolutionNo, FieldName,
       RealDouble, RangeMin, RangeMax,
       Values, ier)
 enddo   ! field loop
enddo    ! solution loop
```

Fig.14 Example Reading Flow solution Data

Each Fortran function holds one more argument than its C counterpart. The error status, `ier`, is the return value of most C routines. Since Fortran subroutines do not allow for return value, the error status is included in the argument list.

The Fortran routine `cg_nsols_f` returns the number of flow solutions recorded for a particular zone of a CGNS database. A zone may hold none to several flow solution sets. Each flow solution set (`SolutionNo`) is qualified by its name (`SolutionName`) and by the grid location of its solution data, such as `Vertex`, `CellCenter`, `EdgeCenter`, etc. These may be extracted using the function `cg_sol_info_f`. The number of solution vectors (`Nfields`) contained in a solution set is obtained with the routine `cg_nfields_f`.

The function `cg_field_info_f` is optional. It determines the name of the solution vector (`FieldName`) if it is not yet known by the application. It also returns the data type (`DataType`) used to stored the solution on disk. The solution vectors are read using the routine `cg_field_read_f`. This routine works the same way as `cg_coord_read` defined in the previous example. It allows setting the name of the variable desired (`FieldName`). It is advised but not mandatory to use the data-name identifiers defined in the SIDS, e.g. `Density`, `Massflow`, etc. The precision of the solution vector returned by the function may be set by the CFD application (`RealDouble` or `RealSingle`) independently of the format used to record these data on disk. As for coordinate vectors, the CGNS API compares the data type on disk with the one requested and automatically accomplishes any necessary conversions. Finally, the solution vectors may be read only partially, within the range prescribed with `RangeMin` and `RangeMax`. This is particularly useful when plotting cross section results, for example.

3.1.3 Write Zone Connectivity and Overset Holes

Three types of block-to-block connectivity are supported by the SIDS and CGNS API: one-to-one abutting, which is also called point matching or C0, abutting with mismatched points and oversets. The grid connectivity information for overset grids must also account for the overset holes within each zone. The C example shown on figure 15 demonstrates how to write the connectivity information for the three types supported, as well as overset hole information, using the CGNS API.

The routine `cg_conn_write` can be used to write any of the three types of block-to-block connectivity for a given zone. The interface may be identified with a name `ConnectName`. The computational mesh indices used in the definition of the zone sub-range may refer to vertices or cells. The variable `GridLocation` holds the

```
for (ConnNo=1; ConnNo<=NConns; ConnNo++)
{
  cg_conn_write(FileNo, BaseNo, ZoneNo,
  ConnectName, GridLocation,ConnectType,
  PointSetType, Npoints, DonorName,
  NpointsDonor, DonorDataType, Points,
  DonorPoints, InterfaceNo);
}
for (HoleNo=1; HoleNo<=Nholes;HoleNo++){
  cg_hole_write(FileNo, BaseNo, ZoneNo,
  HoleName, GridLocation, PointSetType,
  NPoints, Points, HoleNo)
}
```

Fig.15 Example Writing Zone Grid Connectivity

location adopted, `Vertex` or `CellCenter`. The type of connectivity being recorded is specified with `ConnectType` to one of the following name identifiers: `Overset`, `Abutting` or `Abutting1to1`. `DonorName` holds the name of the adjacent zone (donor) interfacing with the current zone (receiver).

The sub-range of nodes or cells on the receiver side may be defined using a range or a discrete list of points or cells. If a range of points or cells is used, the `PointSetType` is set to `PointRange`. When a discrete list of points or cells is used, the `PointSetType` equals `PointList`. The number of points or cells is defined by `Npoints` on the receiver side and `NpointsDonor` on the donor side (adjacent block). For a point set type `PointRange`, this number always equals two. For a point set type `PointList`, it equals the number of points or cells in the point set. The list of points or cells is recorded in `Points` on the receiver side and `DonorPoints` on the donor side. The donor points may only be defined using a `PointList`.

In the particular case of a point-to-point matching between the two zones, the donor points may be expressed with integer values. For all other cases, the donor points are real values comprising the interpolation factors used to locate the receiver points in the donor zone. The variable `DonorDataType` holds the format used to define the donor points. The eligible data types are `Integer`, `RealSingle` and `RealDouble`. This function returns an index for the interface number.

The function `cg_hole_write` writes a new overset hole in an existing zone. The overset hole name is specified by `HoleName`. Its location is defined by a list of indices, which may refer either to vertices or cells location. The grid location is identified with `GridLocation`, and may take the values `Vertex` or `CellCenter`. The extent of the overset hole is specified using one or more range of points or cells (`PointRange`), or with a discrete list of all points or cells in the overset hole (`PointList`). The type of point set used is recorded in `PointSetType`. The number of points or cells

in the point set is defined by `Npoints` and the list of points or cells is recorded in `Points`. This routine returns an index number for the overset hole.

3.2 Applications Supporting CGNS

Several applications have already implemented the CGNS standard successfully. Researchers at NASA Ames have incorporated the CGNS data exchange capability into a developmental version of PEGASUS and OVERFLOW, for fluid dynamics computation on overset grids. Similarly, the CGNS system has been implemented in a research version of the CFL3D solver at NASA Langley, which performs flow computations on multi-block, one-to-one abutting meshes. The ICEM CFD Visual3 post-processor reads CGNS files directly, without the need for data format translation. Additionally, several CFD applications such as Plot3D, NPARC, TLNS3D, ICEM CFD and WIND have translation routines to and/or from the CGNS file standard.

The CGNS system is being released to the public for the purpose of establishing a standard for aerodynamic data storage. Version 1.0 of the CGNS system comprises the ADF and ADF Core software and documentation, the SIDS documentation for multi-block structured CFD analysis, the corresponding "SIDS-to-ADF Mapping" specification, and the CGNS library for most structures defined in the SIDS. A second release is planned for September 1998, which will include support for unstructured and hybrid configurations, as well as geometry-to-mesh association.

4. Conclusion

This paper described the CGNS system, from its original conception to its successful implementation. It has been developed with participation from NASA and US airframe manufacturers to help stabilize the archiving of aerodynamic data. The CGNS system is conceived to support seamless communication of analysis databases between user sites, system architectures, and CFD applications, without concerns for I/O compatibility. It incorporates a set of conventions for the processing and archiving of computational fluid dynamics data, aimed at providing a standard for processing CFD information.

The CGNS system is built over a hierarchical data structure called ADF. The hierarchical quality of this data structure is particularly suited for the storage of CFD information, which is typically composed of a small number of very large arrays. The tree structure may be quickly traversed and sorted without the need of processing irrelevant information. The databases themselves are stored in compact C binary format. They are made machine independent through internal byte ordering translations, performed as needed and invisible to the user or application. Efficient linking capacity allows the partitioning of the data over several files without reducing the performance of the data exchange.

The conventions defined in the SIDS provide for the recording of an extremely complete and flexible problem description. Due to the nature of typical CFD data sets, it is possible to include in the CGNS data structure thorough description of the data with relatively little storage overhead and performance penalty.

The CGNS system supports structured, unstructured and mixed topology, where multi-block connectivity may be either one-to-one abutting, mismatched abutting or overset. A database may contain any number of structured and/or unstructured zones. For unstructured zones, the element connectivity can be stored for a wide range of linear and higher order element shapes. The mesh data is linked to the CAD data within the CGNS system to facilitate quick re-meshing after design changes or mesh optimization.

The flow solutions may be defined at the vertices, or at cell, face or edge centers. Solution vectors are stored using precise naming conventions. Any number of flow variables may be recorded, with or without the use of standardized names. Boundary conditions may be defined on the computational mesh and/or on the CAD geometry, whichever is best suited for a particular CFD application.

The CGNS system also provides for the storage of several types of auxiliary data. This include the conventions for archiving the governing flow equations, the reference state quantities, the convergence history information, any generic discrete or integral data, the dimensional units and exponents, and the nondimensionalization information. User's comments or documentation may be appended nearly anywhere. Site specific data can be incorporated throughout a CGNS database without hindering its compatibility with the CGNS API.

The CGNS system may be implemented in any CFD application by way of a complete and extensible library of functions. The API is platform independent and can be easily implemented in C, C++, Fortran77 and Fortran90 applications. It performs extensive error checking on the database and informs the user of any irregularities via precise error diagnosis messages. Currently the CGNS system is available on most architecture commonly used for CFD analysis: Cray/Unicos, SUN/Solaris, SGI/IRIX, IBM/AIX, HP/UX, DEC-Alpha/OSF. Windows NT support, on Dec and Intel platforms, is planned in the near future. Several applications, in the CFD research community and in industry, have already incorporated the CGNS standard successfully.

The CGNS system is offered to the CFD community for the purpose of establishing a standard for aerodynamic data storage. Documentation, source code, examples of implementation, and compiled libraries are available at www.cgns.org, and online support may be obtained by writing to CGNS-Support@cgns.org. By improving the interoperability of existing and future CFD tools, software development affords the opportunity to focus on functionality and reliability. The CGNS system should lead to the development of shared, reusable software selected on technical merit without concern for I/O compatibility.

The ultimate goal of the CGNS system is to provide a standard designed to satisfy the requirements of the whole CFD community. Consequently, the present and future developments of CGNS are closely tailored to the need of CFD groups in industry, government research centers, and academia. Future projects may include the support of material properties, chemistry data, real gas effects, electromagnetic data, multi-phase flow, etc. The CGNS system could be extended to other types of engineering analysis data, and therefore serve multi-disciplinary applications. Experimental results such as pressure paint data, flight test and wind tunnel measurements could also be incorporated to the SIDS. This would allow a global standard for the recording of both numerical and experimental data. Likewise, advanced diagnosis on CGNS grids would be a handy feature. In addition, the software capability will need to follow the ever-changing trend of the industry. Runtime data management or parallel data handling (MPI) could become a desirable asset for CFD data processing. Similarly, special compression tools may eventually be required to support complex unsteady 3D cases. The future of CGNS resides primarily in the requirements of its users, and the CGNS team engages itself in continuing to serve the needs of the CFD community.

5. Acknowledgments

The CGNS system was developed by the Boeing Commercial Airplane Group under NASA contract NAS1-20267 during the period 1995-98, with extensive participation by a team of scientists and engineers from NASA Langley Research Center, NASA Ames Research Center, NASA Lewis Research Center, McDonnell-Douglas Corporation (now Boeing St-Louis), Arnold Engineering Development Center and ICEM CFD Engineering. The entire CGNS team (about 35 members) deserves credit for the work accomplished, and the project might easily be cited as a prime example of effective voluntary cooperation between NASA and industry.

The authors wish to express the noteworthy contributions of a few individuals who dedicated special effort to this project. Thomas Dickens's software knowledge was valuable in the design and realization of the ADF Core, now maintained by Dan Owen. Ben Paul secured the initial funding and shepherded the project through most of the contract. Wayne Jones contributions were invaluable for testing the CGNS abstractions in real code, helping in the development of the ADF Core and creating the first high level functions. Chuck Keagle designed and executed careful testing procedures for the ADF Core. Gary Shurtleff wrote the TLNS3D and NPARC prototypes, which were the first examples of working CGNS software. Special thanks go to Chris Rumsey and Cetin Kiris's pioneering efforts to implement the CGNS API in NASA solvers. Ray Cosner was a reliable supporter who helped move things forward when progress slowed. Mark Fisher's expertise in data management brought insightful suggestions for the design of the CGNS API. Finally, thanks to Susan Jacob's initial guidance, which helped move the team together in the same direction.

References

[1] CGNS Team, "The CGNS System Overview and Entry Level Document", Draft, Version 1.0, May 1998.

[2] Allmaras, S., "CGNS Standard Interface Data Structures", Draft, May 1997.

[3] CGNS Team, "The ADF User's Guide", May 1997.

[4] CGNS Team, "CGNS File Mapping Manual", Draft, October 1996.

[5] Poirier, D., "CGNS I/O Library", Draft, September 1997.

[6] Coirier, W.J., Golos, F.N., Harrand, V.J., Przekwas, A.J., "CFD-DTF: A Data Transfer Facility for CFD and Multi-Disciplinary Analyses", *36th Aerospace Sciences Meeting & Exhibit*, AIAA-98-0125, Reno, NV, January 1998.

[7] "Getting Started with HDF", National Center for Supercomputing Applications (NCSA), University of Illinois, Chicago, Illinois, May 1993.

[8] "Common File Format (CFF) Programmers Guide", McDonnell-Douglas Aerospace Corporation, St-Louis Missouri, December 1992.

[9] Rew, R. K. and G. P. Davis, "The Unidata netCDF: Software for Scientific Data Access," *Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, Anaheim, California, American Meteorology Society, February 1990.

[10] "Common File Programmers Guide", The Boeing Company, St-Louis Missouri, February 1998.

[11] "The WIND Code User's Guide", The NPARC Alliance, Draft, October 1997.